Runtime Tags Aren't Necessary

Andrew W. Appel*
Princeton University

CS-TR-142-88

March 1988

Abstract

Many modern programming environments use tag bits at runtime to distinguish objects of different types. This is particularly common in systems with garbage collection, since the garbage collector must be able to distinguish pointers from non-pointers, and to learn the length of records pointed to.

The use of tag bits leads to inefficiency. They take up memory (though generally not too much); but more important, tag bits must be stripped off of data before arithmetic operations are performed, and re-attached to the data when it is stored into memory. This takes either extra instructions at runtime, or special tag-handling hardware, or both.

This paper shows how the use of tag bits, record descriptor words, explicit type parameters, and the like can be avoided in languages (like ML) with static polymorphic typechecking. Though a form of tag will still be required for user-defined variant records, all other type information can be encoded once—in the program— rather than replicated many times in the data. This can lead to savings both in space and time.

*Supported in part by NSF Grant DCR-8603453 and by a Digital Equipment Corp. Faculty Incentive Grant

1 Determining types at runtime

In any system with automatic garbage collection, the garbage collector must be able to determine some information about the types of objects at runtime. A pointer must be traversed by the collector; but an integer or other "leaf" does not require any further processing. The garbage collector will need to know the size of each record, and which of the fields are pointers.

These problems have been solved in two different ways. In the earliest LISP systems, one area of memory was reserved for `cons` cells, and another area was reserved for integers. Thus, a machine word containing the address of a `cons` cell is distinguishable from a word containing the address of an integer by the numerical value of the address. All `cons` cells are the same size, so the size of a cell is always known. Encoding types by address ranges has been generalized in the BIBOP (Big Bag of Pages) scheme, in which a table maps address ranges to types.

Pure address-range schemes have the disadvantage that integers now require two words for their storage: one word for the pointer into the "integer area," and one word within that area for the integer itself. Not only does this take extra space, but now an extra fetch is required whenever arithmetic is to be performed on the integer. This scheme was reasonable in early implementations where pointers were half the size of integers, but it is less attractive now.

Another approach is to reserve one or more bits in each word of memory as a "tag," indicating the type of the word. For example, there might be one bit that distinguishes pointers from integers. Using a "tag bit" has two disadvantages:

1. The representable range of addresses and integers is restricted.
2. Addresses and integers now require special handling at runtime to remove tag bits before operations are performed on them, and to apply tag bits before they are stored into memory.

Tag-bits can be handled by special hardware; but special hardware is more expensive and tends to be available in slower realizations than general-purpose machines with larger markets.

Ungar [10] has a good survey of tagging schemes.

2 Statically-typed languages

Many programming languages have static (compile-time) type checking. In such languages, tags are not necessary for normal execution of the program. However, in some implementations of statically-typed languages with garbage collection [8] a descriptor is attached to the beginning of each record for use by the garbage collector. The descriptor specifies the length of the record, and identifies which fields are pointers and which are not.

It is not necessary to put a descriptor on each record in an Algol-like statically-typed language. Instead, a digested version of the user's type definitions, along with a static summary of the type of each variable and procedure parameter, can be provided to the garbage collector[4]. Because this information can be formed (by the compiler) from the program, regardless of the data the program manipulates, it is of a fixed (small) size and requires no runtime overhead for its manipulation.

When the garbage collector is invoked, it searches the stack for references into the heap. From the return-address information on the stack, it can determine which procedure is associated with each stack frame. From the saved type information, the garbage collector can determine the type and value of each local variable on the stack, as well as the types and values of global variables.

The garbage collector then traverses the heap, starting from the roots it finds in global variables and on the stack. As it makes its traversal, it keeps track of the type of each pointer. When following a pointer in the collection algorithm, the collector also makes the appropriate traversal of the type information associated with each pointer. For example, when examining the third field of the record pointed to by P, the collector will examine the (static) type information of P to determine the type of the third field pointed to.

Variant records pose a slight problem. In many languages (like Pascal and ML), variant records have a user-accessible field that determines the type of the record. Consider the Pascal record

```
  TYPE vehicle = RECORD
  CASE kind : (car, truck)
   OF  car : (passengers : integer);
      truck: (weight : integer)
 END
```

This record requires the `kind` field to distinguish whether it is a car or a truck.

The garbage collector must examine the variant field to determine which variant has been instantiated; it will then use this value to find the appropriate type in the static type information. The variant tag must be present in the data structure, however. In Pascal (for example) there is a version of variant records without an explicitly named tag field, but the compiler must insert and initialize such a field at runtime.

The variant field associated with variant records are not strictly comparable with tags as described in section 1. Most records are not variant, and thus won't need a variant field. Furthermore, they serve as user data as well as type information. For these reasons, we will represent them explicitly and still claim to have no "tag" storage.

3 Garbage collection cost *vs.* execution cost

The previous section (and reference [4]) show describe a runtime system for Pascal that has no tags except for variant records and represents integers in their natural (tagless) representation; call this system A. Let us compare its cost to a (hypothetical) Pascal system, B, that uses tags to tell the garbage collector about the types of objects; and to a conventional Pascal system (C) without garbage collection that uses the `dispose` command to do explicit deallocation.

We will assume that there is a performance penalty in system B of $b\%$ for manipulating tags bits at run-time and for creating the record tag words. We will also assume that program C eventually disposes of each cell, with a cost of c instructions per cell.[1] Finally, we suppose that system A's garbage collector is a times as slow as system B's, since it has to interpret the type map as well as traverse the heap.

No matter what the values of a, b, and c, it turns out that system A will be faster in the limit of large memory size. Let L be the number of live records in the heap; this is independent of the program and of the compiler. Let M be the size of memory, and k be the time to copy and scan one record. Then the cost-per-cell-freed of copying garbage collection is

$$\frac{\text{cost of copying}}{\text{cells freed}} = \frac{kL}{\frac{M}{2} - L}$$

which approaches zero[2] as M increases[1]. If we multiply this cost by the constant a, it still approaches zero. On the other hand, the costs b and c of manipulating tags and doing explicit deallocations do not disappear. So it appears that tagless garbage collection is a good thing.

4 Polymorphic type checking

Tagless type checking serves Pascal very well. However, the strict type system of Pascal (and similar languages) constrains the style of programming. To do list processing in Pascal, for example, it is necessary to re-implement the same set of procedures (`cons, head, tail, map, append,` etc.) for each data type of which it is desired to have lists. In contrast, the language LISP has just one set of list primitives which are applicable to lists of any type. This simplifies and clarifies programs, and permits a more abstract programming style.

Languages like ML[6] have a static type system that allows functions (like `cons`) to be polymorphic[7]. That is, the language is type-checked at compile time, but the cons function can be used to build lists of any type of object.

[1] We will ignore the very significant, but harder to measure, programming overhead of using a system without automatic garbage collection.

[2] The cost approaches zero even faster if a generational garbage collector [5][9] is used.

For example, the simplest polymorphic function is the identity function, $(\lambda x.x)$, which has the type $\forall\alpha.(\alpha \to \alpha)$. The function-composition function $(\lambda(f, g).\lambda x.f(gx))$ has the type $\forall\alpha\forall\beta\forall\gamma.((\beta \to \gamma) \times (\alpha \to \beta)) \to (\alpha \to \gamma))$.

Consider the *cons* function that attaches an integer to the beginning of a list of integers, or a string to the beginning of a list of strings. In general, *cons* takes an argument of type α and an argument of type αlist, and returns a result of type αlist, for any type α. Formally, we write the type of cons as $\forall\alpha.(\alpha \times \alpha\text{list} \to \alpha\text{list})$.

The empty list, `nil`, is also polymorphic; it is simultaneously an empty list of integers, and empty list of strings, etc. Its type is $\forall\alpha.\alpha$list. Then the following expressions have the noted types:

`cons(5,nil)`	int list
`cons("s",nil)`	string list
`cons(8,cons(5,nil))`	int list
`cons(cons(5,nil),cons(6,nil),nil)`	(int list) list
`cons(nil,nil)`	$\forall\alpha.(\alpha\text{list})\text{list}$
`cons("s",cons(5,nil))`	illegal

To make a list containing objects of different types, one would use a union type (like a variant record in Pascal).

In ML (as in Lisp), the same piece of machine code implements the *cons* function for integers and the *cons* function for strings. All objects, whether integers or pointers, take exactly one word to represent, and the *cons* function just combines the two one-word arguments into a record. In general, a polymorphic function needs only one implementation.

Other operations (like `plus`) are *overloaded* not polymorphic; their implementation depends on the type of the operand. In ML (but not in Lisp), the compiler determines the argument type at compile-time, and no tag-checking is required at run-time.

No runtime operations except garbage collection depend on the tags in an ML system. If garbage collection could be done without tags, then the tags could be eliminated entirely.

5 Polymorphic garbage collection without tags

Section 3 describes an algorithm for garbage collection of statically-typed, non-polymorphic languages without tags. We would like to extend this scheme to polymorphic languages.

When the garbage collector is invoked, it examines the previously executing function-call frame to see what function was interrupted. This can be determined from the program counter in that frame. As in described in section 3, the machine code for each function can be annotated with a description of the types of the variables in the frame.

The problem is that some of the variables may be polymorphic. Suppose it is the cons function that runs out of memory; its two arguments have type α and αlist, respectively. From this we know that the second argument is either nil (perhaps represented by 0) or a two-element record; but we don't know much about the first argument, and we don't know what type of object is in the record.

The answer turns out to be easy, in principle. We know that cons was called from some other function, and perhaps that function knows what type α is. We can find the calling function by examining the return address in cons's call frame. For example:

```
fun  cons(a,b) =   ( . . . )


fun f(j : int) = g(cons("abc",nil), cons(j+4, nil))
```

The function f calls cons twice; in the first call α is bound to the type *string*, and in the second call α is *int*. If this information is recorded in a data structure attached to the machine code for f, then the garbage collector can find the binding for α by looking at the return address of the cons function. (It can distinguish one call from another because the return addresses will differ.)

What if the calling function were also polymorphic? Here is an example of such a function:

```
fun repeat(d,i) = if i=0 then nil else cons(d, repeat(d, i-1))
```

The repeat function has type $\forall\beta.(\beta \times \text{int}) \rightarrow (\beta\text{list})$. The compiler can unify β with the α used in this call to cons; that is, it can statically determine that the type of the first argument to cons is the same as the type of the first argument to repeat. But how is the garbage collector to know what β is?

Well, repeat must have been called from somewhere. By induction, the garbage collector can keep unwinding the runtime stack until it finds the bindings of all variables. The induction is well-founded (the unwinding terminates) because the top-level expression, which makes the first function call, does not have a polymorphic type.

This will determine the types of all variables in function-call frames. These variables serve as the roots of the reachable graph of data.

The collector can do a depth-first traversal of this graph, keeping track of the type of each record. When the exact type of a given record is known, then it is easy to derive the types of the records it points to. Thus, a depth-first copying collector can be implemented.

6 Breadth-first copying

Most copying garbage collectors use breadth-first traversal, mostly because it is simpler to implement. Some specialized algorithms [2] require a "random-access" breadth-first traversal. Tagless collection is more naturally depth-first, but it is possible to do a breadth-first traversal.

A breadth-first copying collector copies cells from a "from-space" to a "to-space." As each cell is copied, the first word of the cell in from-space is overwritten with a pointer to the copy in to-space. The queue needed for breadth-first search is actually the to-space itself; it is scanned for more pointers to cells that need copying.

The modification to the standard copying algorithm is that a type tag will be added to each cell as it is copied. The collector starts with a set of "roots"—e.g. the contents of procedure-call frames. Any cells reachable from these roots are to be copied, and as explained in the previous section, the collector knows the exact type of each root. Whenever a cell is copied, a pointer to its type is stored with the copy in the to-space. When a cell in to-space is scanned to see if its fields point into from-space, the type-pointer can be used to learn the types of the fields.

The type-pointer need not be kept in the to-space itself; a separate queue of type-pointers will suffice. This has the advantage that after collection the queue may be thrown away.

7 Generational garbage collection

For languages that rarely alter previously-allocated cells, generational garbage collection[5][9] is most efficient. In this scheme, memory is grouped into several regions containing cells of various ages. Cells in newer regions may point to cells in older regions, but not vice versa. Garbage collection can be done mostly on the new (most volatile) regions, and more rarely on the older regions.

The only way that an older region can point to a newer region is if a cell in the older region is altered, which by assumption is rare. When this happens, the older cell must be considered a root for garbage collection of the newer region, and the address of the older cell is put onto a list of such roots to be processed by the collector.

The tagless garbage collector will need to know the type of the older cell (and, in particular, of the newer cell it points to). The older cell must have been copied during a previous collection, however, and the garbage collector knew its type at that time. We could arrange that each copied cell keeps its type-pointer (as described in the last section).

Keeping the type-pointers of all copied cells becomes expensive, however—it will take as much space[3] as an implementation with explicit descriptors everywhere. In ML, however, the compiler can identify the cells that may be stored into; such cells are marked by the programmer as `ref` variables. The garbage collector can attach type-pointer to `ref` cells when it copies them, and omit the type-pointer from other cells.

[3]But not as much time, because most cells become garbage and never get copied.

8 Estimating the performance

Though we have not implemented this algorithm, we can estimate its performance relative to ordinary (tagged) garbage collection. We will make some arbitrary guesses about the cost of various operations, and calculate the resulting impact on time and space of an executing program.

We will use the Standard ML of New Jersey compiler[3] for comparison. This compiler uses a low-order tag bit on each word to distinguish pointers from integers, and uses a one-word descriptor on each record to communicate its type to the garbage collector. The tag bits and descriptor words are used only by the garbage collector. [4]

We ran a large benchmark in this system (the compiler compiling itself; 500 seconds of execution on a VAX 8550, or approximately $2.7 \cdot 10^9$ instructions assuming 5 MIPS). We used a generational garbage collector; the cost of garbage collection decreases as one increases the memory limit, but our typical overhead in large memories is one or two percent.

The benchmark allocated 31.4 million records of average size 2.9 words. This means that the space cost of the descriptors is about 26%. On the average, a record is allocated every 88 instructions; so the overhead of the *store* instructions to create the descriptors is about 1.1%.

It is harder to estimate the cost of the tag bits. Some compilers reserve the low-order bit of each word to distinguish integers from pointers. On byte-addressable machines, pointers don't really need this bit anyway; and integers can be shifted left from their natural representation. Integer addition can then be done with the ordinary machine add instruction, and no shifting or correction will be necessary (since $2x + 2y = 2(x + y)$). We prefer that pointers have their "natural" machine representation, and we use a low-order tag of one for integers. This requires a one-instruction correction to one of the arguments of an add or subtract, and three extra instructions to correct a multiply. Addition or subtraction of a constant, however, can be corrected at compile-time, and comparison operations require no correction. On the whole we will arbitrarily assume that tagged arithmetic takes 30% longer than untagged arithmetic.

Ungar[10] measured the dynamic frequency of tagged arithmetic instructions in Smalltalk and found them to be approximately 9% of all instructions. Thus, there is an estimated 3% time overhead from tag bits.

The space overhead from tagging is, presumably, 1 bit for every integer stored. When a small integer is represented, that bit isn't needed anyway. And when a full 32-bit integer is represented, then it just won't fit. 32-bit integers are often needed in multi-precision arithmetic packages, to represent machine instructions, etc. The tag bit, therefore, is more of an unquantifiable inconvenience than a measurable cost.

In total, the overhead of descriptors and tag bits is about 25% in space and

[4]and by the polymorphic equality feature, which is dispensible.

4% in time. By using BIBOP schemes, we can reduce the space overhead but not the time, as allocating a record then becomes more complicated.

In contrast, let us estimate the overhead of tagless garbage collection. The garbage collector must do an extra traversal of the stack to determine the instantiations of polymorphic types; but this is trivial in comparison with the amount of data to be copied. Copying might become slower by a factor of two or three, since the type system must be traversed at the same time. There is no other time cost; so the overhead would increase by an amount equal to perhaps twice the current cost of garbage collection, i.e. 2–4%.

There is a space cost of tagless garbage collection, too. The machine code for each function must be annotated with descriptions of the types of its local variables and of the functions it calls. This might be half the size of the executable code. In our benchmark, code was usually just under half of all live data at any time, so the effective space overhead of tagless collection might be on the order of 25%. For programs that have much more data than code (and such programs are common), the space overhead vanishes.

So we find that the cost of tagless collection would probably be about equal to the cost of conventional garbage collection, in the given benchmark. For machines with more memory (where the overhead of garbage collection decreases), and for programs with much more data than program, the cost of tagless collection will begin to win.

9 Conclusion

Run-time tags are not necessary for statically-typed polymorphic languages. The performance of a system without type tags will probably be comparable the the performance of a conventional system. Programming language features (e.g. the polymorphic equality feature of Standard ML) that require runtime tags should not be thrown into the language "because the tags have to be there anyway;" the tags don't have to be there.

Acknowledgement

Thanks to Trevor Jim for many useful comments on the manuscript.

References

[1] Andrew W. Appel. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

[2] Andrew W. Appel, John R. Ellis, and Kai Li. *Real-time Concurrent Collection on Stock Multiprocessors.* Technical Report CSL-TR-133-88, Princeton University, 1988.

[3] Andrew W. Appel and David B. MacQueen. A Standard ML compiler. In Gilles Kahn, editor, *Functional Programming Languages and Computer Architecture (LNCS 274)*, pages 301–324, Springer–Verlag, 1987.

[4] Dianne E. Britton. *Heap Storage Management for the Programming Language Pascal.* Master's thesis, University of Arizona, 1975.

[5] Henry Lieberman and Carl Hewitt. A real-time garbage collector based on the lifetimes of objects. *Communications of the ACM*, 23(6):419–429, 1983.

[6] Robin Milner. A proposal for Standard ML. In *ACM Symposium on LISP and Functional Programming*, pages 184–197, 1984.

[7] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[8] Paul Rovner, Roy Levin, and John Wick. *On Extending Modula-2 For Building Large, Integrated Systems.* Technical Report Research Report 3, DEC Systems Research Center, Palo Alto, CA, 1985.

[9] David Ungar. Generation scavenging: a non-disruptive high performance storage reclamation algorithm. In *SIGPLAN Notices (Proc. ACM SIGSOFT/SIGPLAN Software Eng. Symp. on Practical Software Development Environments)*, pages 157–167, 1984.

[10] David M. Ungar. *The Design and Evaluation of a High Performance Smalltalk System.* MIT Press, Cambridge, Mass., 1986.